# Document made available under the Patent Cooperation Treaty (PCT)

International application number: PCT/CA05/000194

International filing date:      16 February 2005 (16.02.2005)


Document type:     Certified copy of priority document

Document details:    Country/Office: CA
                      Number:       2,479,485
                      Filing date:    20 September 2004 (20.09.2004)


Date of receipt at the International Bureau:   06 April 2005 (06.04.2005)


Remark:   Priority document submitted or transmitted to the International Bureau in compliance with Rule 17.1(a) or (b)

**Office de la propriété intellectuelle du Canada**

Un organisme d'Industrie Canada

**Canadian Intellectual Property Office**

An Agency of Industry Canada

**PCT/CA** 2 0 0 5 / 0 0 0 1 9 4

**15 MARCH 2005** 15·03.05

*Bureau canadien des brevets*

Certification

*Canadian Patent Office*

Certification

La présente atteste que les documents ci-joints, dont la liste figure ci-dessous, sont des copies authentiques des documents déposés au Bureau des brevets.

This is to certify that the documents attached hereto and identified below are true copies of the documents on file in the Patent Office.

Specification, as originally filed, with Application for Patent Serial No: **2,479,485,** on September 20, 2004, by **CHRIS DAVIES,** for "System and Method for a Self-Organizing, Reliable, Scalable Network".

_____
Agent certificateur/Certifying Officer

March 15, 2005
_____
Date

**Canada**

(CIPO 68)
31-03-04

OPIC CIPO

# System and Method for a Self-Organizing, Reliable, Scalable Network

## Abstract

This network system enables individual nodes in the network to coordinate their activities such that the sum of their activities allows communication between nodes in the network.

This network is similar to existing ad-hoc networks used in a wireless environment. The principle limitation of those networks is the ability to scale past a few hundred nodes. This method overcomes that scaling problem.

**Note to Readers**: examples are given throughout this document in order to clarify understanding. These examples, when making specific reference to numbers, other parties' software or other specifics, are not meant to limit the generality of the method and system described herein

## Nodes

Each node in this network is directly connected to one or more other nodes. A node could be a computer, network adapter, switch, or any device that contains memory and ability to process data. Each node has no knowledge of other nodes except those nodes to which it is directly connected. A connection between two nodes could be several different connections that are 'bonded' together. The connection could be physical (wires, etc), actual physical items (such as boxes, widgets, liquids, etc), computer buses, radio, microwave, light, quantum interactions, etc.

No limitation on the form of connection is implied by the inventors.
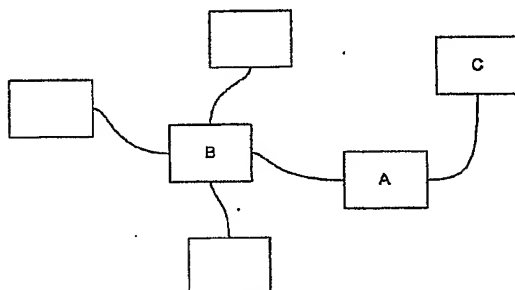
*Node A is directly connected to nodes B and C*

*Node C is only connected to Node A*
*Node B is directly connected to four nodes*

'target nodes' are a subset of all directly connected nodes. Only 'target nodes' will ever be used as routes for messages sent to a particular destination node (discussed later).

## Nodes and Messages

A node must have a globally unique identifier (GUID). This GUID is used to identify the node on the network and allow other nodes to route packets to this node. If a node does not need to have packets routed to it, it does not need a GUID. Nodes without a GUID are still able to forward packets to other nodes.

A node may keep this GUID permanently, or may generate a new GUID on startup. Node A can only send a message to node B if Node A knows the GUID of node B. A node may have multiple GUID's in order to emulate several different nodes.

For the sake of clarity in this document we assume that each node has only one GUID associated with it. This should not be seen as limiting the scope of this invention.

Messages are sent to a certain GUID (that represents a machine), and to a particular port number on that machine (similar to TCP/IP). We refer to the machine where messages are sent to as the destination node. This is to separate it from the 'target node' that is a directly connected node that is the next best step towards the destination node.

Ports are discussed as a destination for messages, however the use of ports in these examples is not meant to limit the invention to only using ports. A person skilled in the art would be aware of other mechanisms that could be used as message destinations. For example, nodes could generate a unique GUID for each connection.

If a node knows about a destination node it will tell those nodes it is connected to about that node. (discussed in detail later). The only node that knows the final destination for a message is the node that is final destination for that message. A node never knows if the node it passes a message to is the ultimate destination for that message.

A destination node is the ultimate destination for a message.

For example, if node N tells a directly connected node about destination node N it is telling that directly connected node about itself.

2

At no point does any node attempt to build network map of any sort (global or local), or have any knowledge of the network as a whole except of the nodes it is directly connected to.

## Initial Connection

When a node first connects to another node in the network it tells the other node three things:

1. A Tie Breaker Number
   This is a very large random number used as a tie-breaker, in case there are two equal node choices, or both nodes choose each other as 'target nodes' for the same destination node at the same time.
2. Destination Node Count
   Tells the other node the maximum number of destination nodes that this node wants to know about. This is used to ensure that this node is not overrun with node updates that would exceed its available memory.
3. The Connection Cost
   Both nodes must agree on the same connection cost for the connection. This connection cost can be assigned by an operator, determined by examining the type of network adapter, or dynamically discovered before this initial connection message is exchanged. The discovery process can involve the two nodes exchanging data and timing the connection, or agreeing on the physical connection properties.

   Each node will randomly alter this agreed upon connection cost by a small amount that should not exceed 1% of the connection cost. The modified value is sent as the connection cost in this message.

   The node with the highest tie-breaker value will have its connection cost used as the connection cost for the connection by both nodes.

   This connection cost can be re-negotiated at any time during the connection. (discussed later)

Even though this message is sent right at the start, it can be sent later as well in order to reduce or increase the destination node counts.

It is important that each node tells the same tie-breaker number to every directly connected node.

If a node needs to adjust the 'destination node count' number or the connection cost it should send the same 'tie-breaker' number.

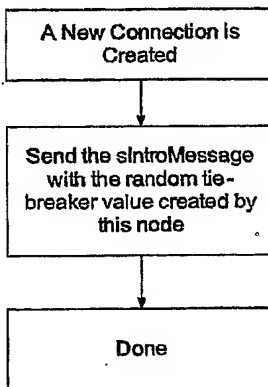The structure used looks like this:

```
struct sIntroMessage {
```

3

```
            // the number used to break ties
            int            uiTieBreakerNumber;

            // the number of destination nodes this node wants to know about.
            int            uiDestinationNodeCount;

            // the connection cost
            float          fConnectionCost;
}
```
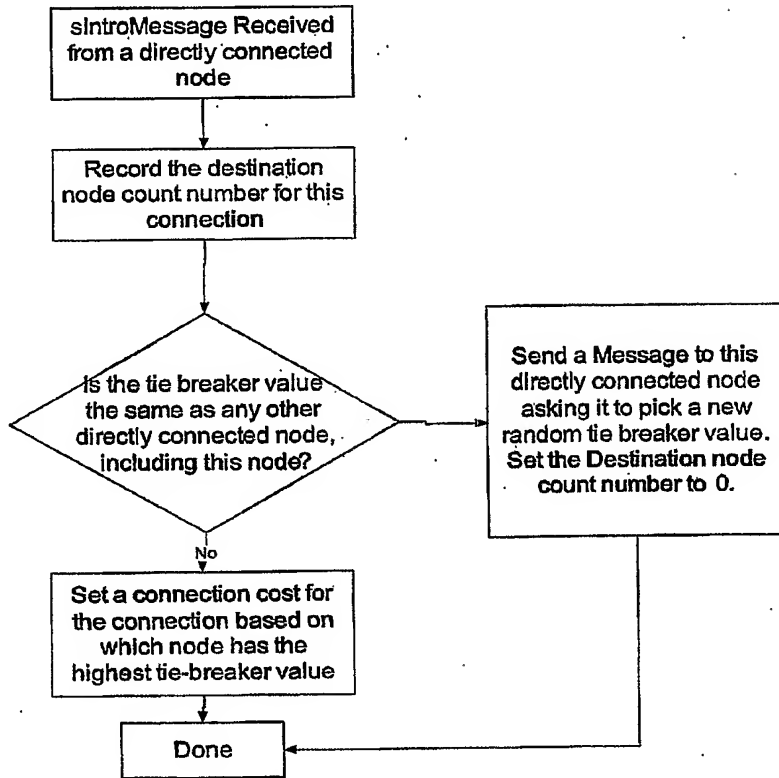
Below is a flowchart of the initialization process after a connection has been established:

```
            +-----------------------------+
            |   A New Connection is       |
            |        Created              |
            +-----------------------------+
                          |
                          v
            +-----------------------------+
            |   Send the sIntroMessage    |
            |   with the random tie-      |
            |   breaker value created by  |
            |   this node                 |
            +-----------------------------+
                          |
                          v
            +-----------------------------+
            |          Done               |
            +-----------------------------+
```

Below is a flowchart of what happens when an sIntroMessage is received from the directly connected node.

4

```
┌─────────────────────────┐
│ sIntroMessage Received  │
│ from a directly connected│
│         node            │
└─────────────────────────┘
            │
            ▼
┌─────────────────────────┐
│  Record the destination │
│ node count number for this│
│       connection        │
└─────────────────────────┘
            │
            ▼
        ◇ Is the tie breaker value        ┌──────────────────────────┐
        the same as any other      ──────▶│  Send a Message to this   │
        directly connected node,          │  directly connected node  │
        including this node? ◇             │   asking it to pick a new │
            │                              │ random tie breaker value. │
           No                             │  Set the Destination node │
            ▼                              │    count number to 0.     │
┌─────────────────────────┐               └──────────────────────────┘
│ Set a connection cost for│                          │
│ the connection based on │                          │
│   which node has the    │                          │
│ highest tie-breaker value│                          │
└─────────────────────────┘                          │
            │                                         │
            ▼                                         │
        ┌─────────┐                                   │
        │  Done   │◀──────────────────────────────────┘
        └─────────┘
```

It is important that a node tells the same tie breaker value to all directly connected nodes.

The structure used to request another random tie-breaker value looks like this:

```
struct sRequestNewTieBreaker {

        // This structure is empty, if the node sees this message it will
        // generate a new tie breaker value and tell all its directly connected
        // nodes this new value

}
```
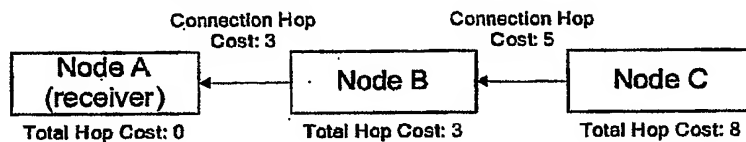
## Calculation Of Hop Cost.

S

'Hop Cost' is an arbitrary value that allows the comparison of two or more routes. In this document the lower the 'hop cost' the better the route. This is a standard approach, and someone skilled in the art will be aware of possible variations.

Hop Cost is a value that is used to describe the capacity or speed of the connection. It is an arbitrary value, and should not change in response to load. It can change in response to the stability of the connection, or other factors that are not a result of the amount of traffic flowing along the connection.

Hop Cost is additive along a connection.

```
           Connection Hop            Connection Hop
              Cost: 3                    Cost 5
      ┌──────────────┐         ┌──────────────┐         ┌──────────────┐
      │   Node A     │◄────────│    Node B    │◄────────│    Node C    │
      │  (receiver)  │         │              │         │              │
      └──────────────┘         └──────────────┘         └──────────────┘
      Total Hop Cost: 0        Total Hop Cost: 3        Total Hop Cost: 8
```

In this example, node C has a total hop cost of 8 to reach node A, since connections between node A and B and node B and C total 8.

A lower hop cost should represent a higher capacity connection, a faster connection or a more stable connection. These hop costs will be used to find the best path through the network using a Dykstra like algorithm.

Both nodes must agree on the same hop cost for a given connection. The hop cost that is decided on must have a random variation (for example 1%).

Pipes with low capacity should be assigned a high hop cost. Those with high capacity should be assigned a low hop cost. A high capacity pipe that is not stable should be assigned a higher hop cost then a pipe with the same capacity that is more stable.

It is useful that the Hop Cost of a pipe has an approximately direct relationship to its capacity. For example, a 1Mbit pipe would need to have 10 times the Hop Cost of 10Mbit pipe.

The hop cost for the link can be re-negotiated during the course of the connection based on (but not limited to):

1. Line Quality
2. Uptime
3. Connection consistency
4. Latency

These values should be chosen such that a lower value indicates a better connection and a higher value indicates a worse connection. The approach to deciding what value is assigned to what kind of connection should be somewhat consistent across the entire network.

When the hop cost (or connection cost) for the link has been renegotiated all the destination node updates should be adjusted to reflect the difference between the previous hop cost and new hop cost.

For example, if the old hop cost was 10 and the new hop cost is 12, then 2 should be added to the fHopCostFromFlow and fHopCost for each destination node update from that directly connected node. The choice of which directly connected node should be the target node for each known destination node should also be re-evaluated.

Destination node updates need to be sent to directly connected nodes based on these changes. As always these updates should be sent in order (discussed later).

Someone skilled in the art will be able to assign hop costs, or create a dynamic discovery mechanism.


## End User Software

This network system and method can be used to emulate most other network protocols, or as a base for an entirely new network protocol. In this document TCP/IP will be used as an example of a protocol that can be emulated. The use of TCP/IP as an example is not meant to limit the application of this invention to TCP/IP.

In TCP/IP when a node is turned on, it does not announce its presence to the network. It does not need to because the name of the node (IP address) determines its location. With this new network invention, the node needs the network to know that it exists, and provide the network with a guaranteed path to itself. This is discussed in much greater detail later.

When end user software (EUS) wishes to establish a connection, it can do so in a manner very similar to TCP/IP. In TCP/IP the connection code looks similar to this:

    SOCKET  sNewSocket = Connect(IP Address, port);

With this approach, the 'IP Address' is replaced with a GUID.

    SOCKET sNewSocket = Connect(GUID,port).

In fact, if the IP Address can be guaranteed to be unique, then the IP address could serve as the GUID, providing a seamless replacement of an existing TCP/IP network stack with this new network invention.

7

One way to guarantee a unique IP is to have each node create a random GUID and then use that to communicate with a DHCP like server to request a unique IP address that can be used as a GUID. The node would then discard its first GUID name and use only this IP address as a GUID.

Once the connection has been requested, the network will determine a route to the destination (if such a route exists), and continually improve the route until an optimal route has been found.

The receiving end will look identical to TCP/IP, except a request to determine the IP address of the connecting node will yield a GUID instead. (or an IP address is those are being used as GUIDs).

This approach provides the routing through the network, someone skilled in the art could see how different flow control approaches might work better in different networks. For example, a wireless network might need an approach that does not lose packets when incoming data rates exceed outgoing connection rates.

Below is a diagram indicating where this routing method would fit in the TCP/IP example.
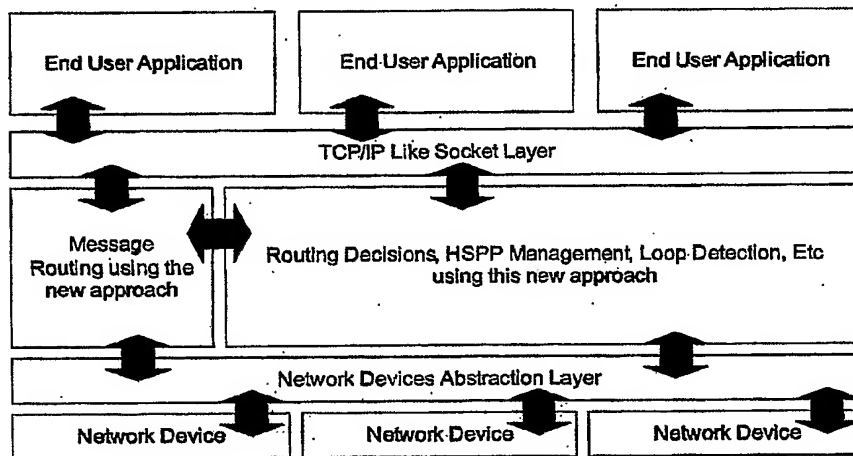


Someone skilled in the art should see that this new routing approach allows a TCP/IP like interface for end user applications. This is an example not meant to limit this routing approach to any particular interface (TCP/IP for example) or application.

## Initial Destination Node Knowledge

When a node is connected to the network, the system needs a way to find a path through the network to this node. This path is through a series of directly

connected nodes. No node is aware of the complete path. Each node is only aware of its next best step to the ultimate destination. It determines this next best step using information provided to it by its directly connected nodes. A node will generally try to minimize its hop cost to the destination node by picking the directly connected node with the lowest hop cost to the destination (discussed in detail later).

When a node is first connected to the network, it tells all directly connected nodes about itself (the exception is if this node is acting as a routing node only). It will also use this same structure to tell directly connected nodes about destination nodes that it has been told about.

1. <u>The name of the Node that this update is about</u>
   This is a name that is unique to the node. Two nodes should never have the same name.
2. <u>Hop Cost</u>
   This is a value that describes how good this route is to this node. Each connection in the path is assigned a 'hop cost', the fHopCost in this message is a reflection of the summation of these individual hop costs between the node that has received this update and the node that this update is about.
3. <u>Distance from data flow</u>
   Discussed Later. Very similar to 'Hop Cost', except that it describes how far this node is from a data flow. This can be used to decide which node updates are 'more important'. A node that is in the data flow has an effective 'fHopCostFromFlow' of 0. The fHopCostFromFlow is a summation of all the hop costs between the node that told this node of this destination node and a data flow for that destination node.
4. <u>Target Node</u>
   We'll tell the node we're sending this update to if it is a 'target' node for messages sent to a destination node. This creates a 'poison reverse'. This poison reverse stop one hop loops from forming, and also plays a roll in removing bad route information from the network.
5. <u>In Data Stream</u>
   If this node is in the data stream, then we'll need to tell our directly connected node that is the 'target node' for this destination node that it is in the data stream. This is discussed in more detail later.
6. <u>Node Weight</u>
   Each node is given a weight; a weight is used to describe the relative size and connectivity of a node. The purpose of assigning a weight is to help the core form near more powerful nodes. It has no impact on data flow between nodes. The weight value needs a relatively small range because it is used as an exponent of 10. For example, a weight of 3 would mean an actual weight of $10^3$ or 1000.

   For example, if there was a mesh network of 10,000 sensors connected to a network with 13 computers, by assigning a weighting 5 to all the computers, it would assure that the core of the network would be created

between the computers, and not between two sensors in the middle of the mesh of 10,000.

This update takes the structure of:

```
struct sDestNodeUpdate {

        // the name of the node. Can be replaced with a number
        // (discussed later)
        sNodeName              nnName;

        // the hop cost this node can provide to the ultimate
        // receiver.
        float                  fHopCost;

        // calculated in a similar fashion to 'fHopCost'.
        // and records the distance from the data flow for this
        // node.
        float                  fHopCostFromFlow;

        // if the node we're sending this to is a target node for this
        // destination node, then we'll set this to true;
        boolean                bIsTargetNode;

        // if the we're in the data stream, and this directly connected
        // node is the target node for this destination node
        boolean                bIsInDataStream;

        // the weight of this node. The weight can use a small range
        // since it is used as an exponent. In an implementation
        // one byte could be used for bIsTargetNode (1 bit) and
        // bIsInDataStream (1 bit) and nWeight (6 bits)
        int                    nWeight;
};
```

Regardless of whether this is a previously unknown destination node or an update to an already known node the same structure is sent.

The receiving node will store this information for as long as it is connected to the directly connected node, or until the fHopCost has been at infinity long enough that it can be removed from memory. (discussed in detail later). Before this information is stored the connection cost for this connection is added to fHopCost and fHopCostFromFlow. If either of these values exceed infinity, they are set to infinity.
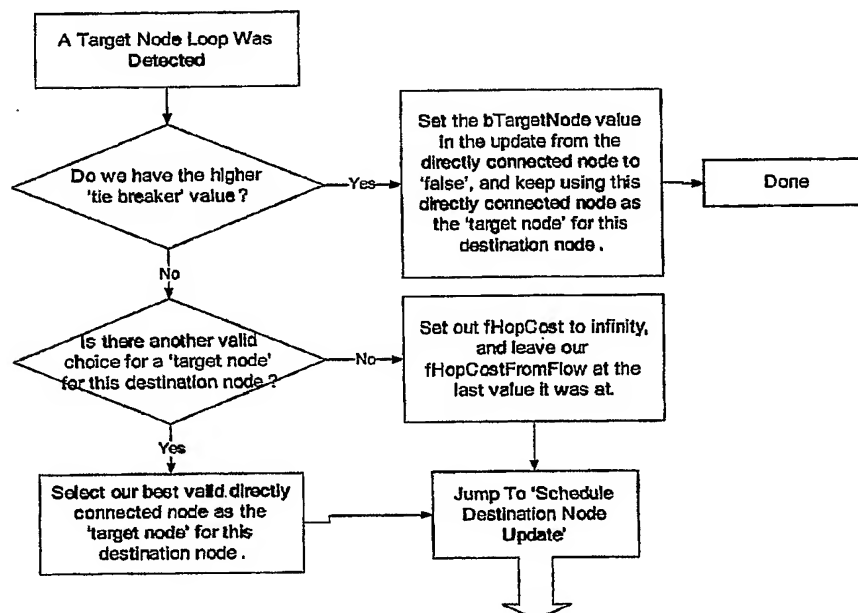
A value of infinity is not a true infinity; it is a value high enough that it should never be reached (for example 9,999,999), but still leaves a large range of values between itself and the maximum value storable by the number). The value of infinity must be the same for all nodes in the network, or the infinity value must be negotiated during the initial connection process.

If this is the first time a directly connected node has heard about this destination node it will choose the directly node that first told it about this destination node as its 'target node' for messages to that destination node. A node will only send EUS messages to a node or nodes that are 'target nodes', even if other nodes tell it that they too provide a route to the destination node.

In this fashion a network is created in which every node is aware of the destination node and has a non-looping route to the destination node through a series of directly connected nodes.

If a node A has chosen node B as a 'target node' for a destination node, node A will tell node B that it is a 'target node' (bIsTargetNode) for that particular destination node. This will create a 'poison reverse' that will stop many loops from being created.

If both nodes tell each other at the same time that they are target nodes, the node with the highest 'tie-breaker' number gets to keep its choice, and the other node must make another choice. The following spread sheet describes this process.

```
┌──────────────────────┐
│ A Target Node Loop Was│
│      Detected         │
└──────────────────────┘
           │
           ▼
      ╱────────────╲          ┌─────────────────────┐       ┌──────────┐
     ╱ Do we have   ╲         │ Set the bTargetNode │       │          │
    ╱ the higher     ╲──Yes──▶│ value in the update │──────▶│   Done   │
    ╲ 'tie breaker'  ╱        │ from the directly   │       │          │
     ╲  value ?     ╱         │ connected node to   │       └──────────┘
      ╲────────────╱          │ 'false', and keep   │
           │                  │ using this directly │
           No                 │ connected node as   │
           │                  │ the 'target node'   │
           ▼                  │ for this destination│
      ╱────────────╲          │       node.         │
     ╱ Is there     ╲         └─────────────────────┘
    ╱ another valid  ╲        ┌─────────────────────┐
    ╲ choice for a    ╲──No──▶│ Set out fHopCost to │
     ╲'target node'   ╱       │ infinity, and leave │
      ╲for this dest. ╱       │ our fHopCostFromFlow│
       ╲────────────╱         │ at the last value   │
           │                  │    it was at.       │
           Yes                └─────────────────────┘
           │                             │
           ▼                             ▼
┌──────────────────────┐      ┌─────────────────────┐
│ Select our best valid│      │ Jump To 'Schedule   │
│ directly connected   │─────▶│ Destination Node    │
│ node as the 'target  │      │      Update'        │
│ node' for this       │      └─────────────────────┘
│ destination node.    │               │
└──────────────────────┘               ▼
```

## What A Node Tells Directly Connected Nodes About Itself

When a node tells all directly connected nodes about itself (using sDestNodeUpdate) it will tell them a fHopCost of 0 and a fHopCostFromFlow of 0.1 or some value that is greater then 0 but not substantially so.

The exception is if node B is directly connected to node A, and node B tells node A that it is in the data stream (bIsTargetNode and bIsInDataStream) for messages that are destined for node A, then node A will send node B a sDestNodeUpdate for destination node A (itself) with a fHopCost and fHopCostFromFlow of 0.

When node B tells node A that it is no longer in the data stream, then node A sends node B a sDestNodeUpdate for destination node A with a fHopCost of 0 and a fHopCostFromFlow of 0.1 (for example).

This is part of the process that will stop data from looping in a network when the destination node has been removed.

## The Spread of Destination Node Knowledge

Below is a series of diagrams illustrating the spread of destination node knowledge.
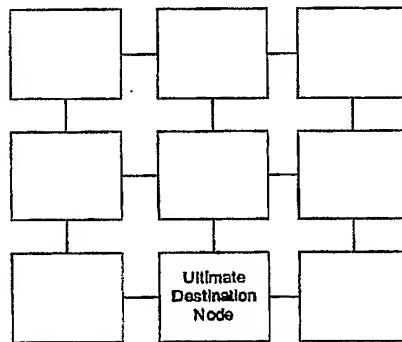


*Step1: The ultimate destination node is connected to several nodes in the network and prepares to send an initial destination node update to its directly connected neighbor nodes.*
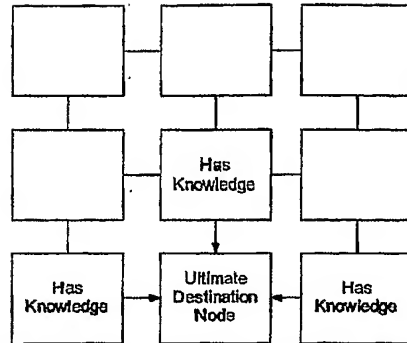
*Step2: As destination node knowledge spreads, each node that gets told of the destination node selects the node that told it as the 'target node' for messages routed to that destination node.*
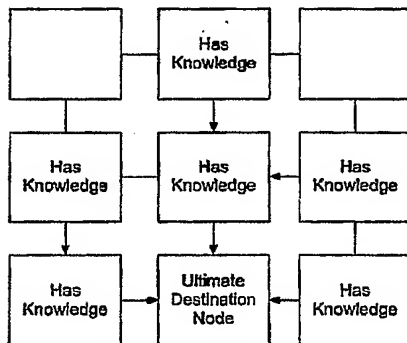


*Step3: Knowledge of the destination node continues to spread, with each node that is told about the destination node choosing the node that told it. If another node provided it with a better hop cost later, the node would choose the directly connected node with the better hop cost.*
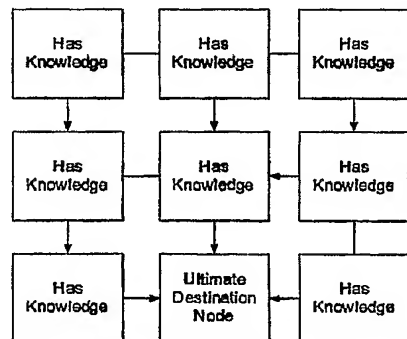


*Step4: In this step all nodes in the network are aware of the destination node and have a 'target node' that they can send messages that are destined for the destination node.*

*Note: the linkages between nodes and the number of nodes in this diagram are exemplar only, whereas in fact there could be indefinite variations of linkages within any network topography, both from any node, between any number of nodes.*

At no point does any node in the network attempt to gather global knowledge of network topology or routes.

Even if a node has multiple possible paths for messages it will only send data messages (as oppose to control messages) to the node that it has chosen as its 'target node'. These data messages are called EUS messages.

A node cannot select as a target node a directly connected node that has already chosen it as a target node.

## Setting the Destination Node Properties

Each node maintains a set of information on the destination nodes it is aware of. The structure for this knowledge looks like this:

```
struct sDestinationNodeInfo {

        // A number that identifies the connection that has
        // been selected as the target node for this destination node
        // it is  -1 if no target node is selected.
        int             nConnectionID;

        // the hop cost to the destination
        float           fHopCost;

        // the hop cost to the flow
        float           fHopCostFromFlow;

        // EUS data message pointers can be stored here as well.

}
```

If there are no valid target nodes, the fHopCost should become infinity, and fHopCostFromFlow should remain unchanged.

A valid target node for a destination node is a directly connected node whose most recent route update for this destination node:

    1. Has a fHopCost that is not infinity
       sDestNodeUpdate.fHopCost < INFINITY

## 2. Has not selected us as a target node.
sDestNodeUpdate.bIsTargetNode == FALSE

Every time a destination node update arrives from a directly connected node we'll decide if there is a better choice for a target node. If there is a better choice we'll select it. We'll pick the valid target node with the lowest fHopCost, if the fHopCosts are tied, we'll pick the directly connected node with the best tie-breaker value as the target node.
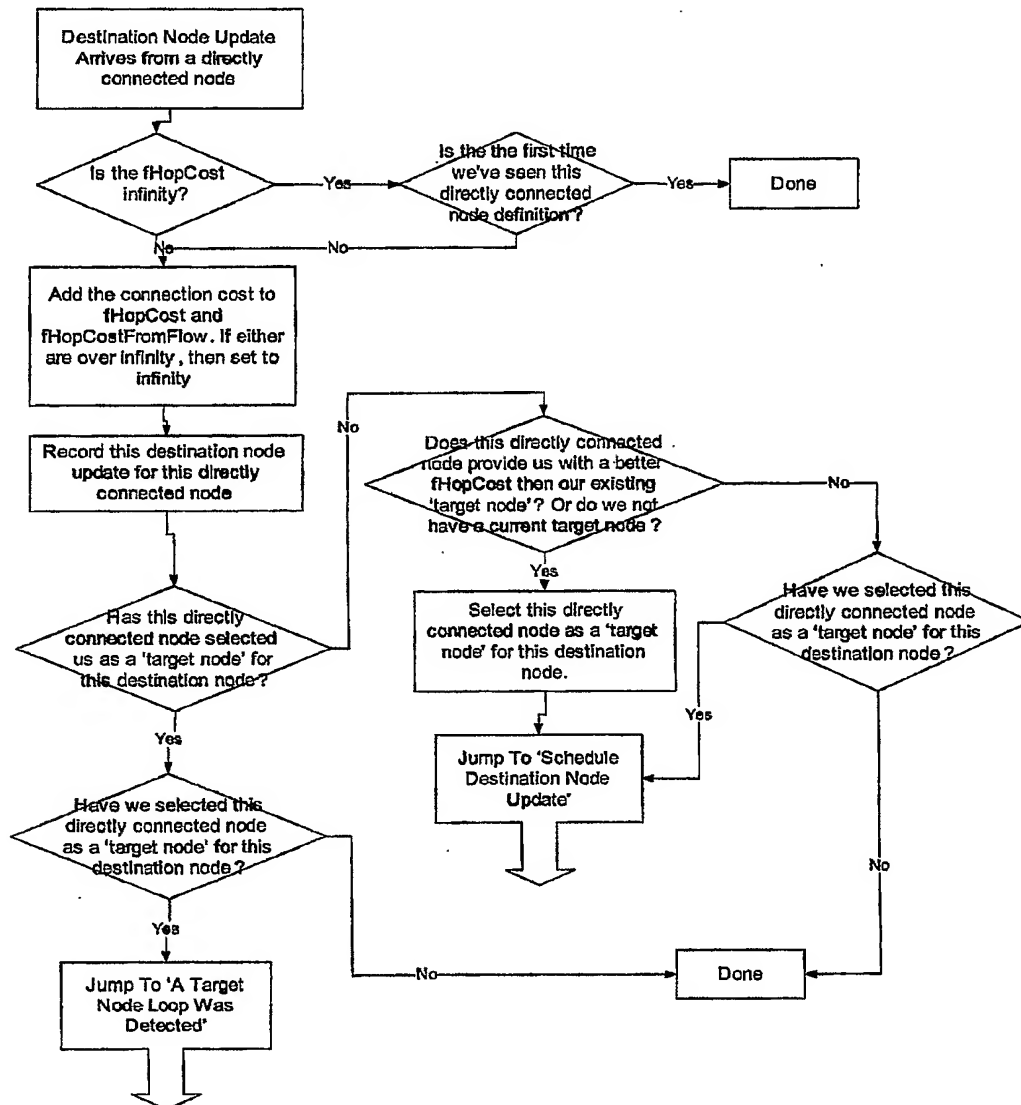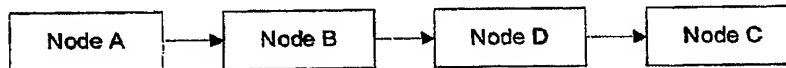
## Nodes In The Data Stream

A node is considered in the data stream if it is on the path for data flowing between an ultimate sender and ultimate receiver.

Being in the data stream is not a requirement for the transfer of EUS or control messages.

A node knows it is in the data stream because a directly connected node that has selected this node as a target node tells it that it is in the data stream. It does this using the standard destination node update structure 'sDestNodeUpdate'.

Node A may only tell node B that it is in the data stream for messages to node C if node A has also told node B that node B is the 'target node' for messages to node C. If node B has been told it is in the data stream for messages to node C, then it will tell its directly connected node (node D) that it has selected as its target node for message to node C, that node D is in the data stream.

```
┌──────────┐      ┌──────────┐      ┌──────────┐      ┌──────────┐
│  Node A  │─────▶│  Node B  │─────▶│  Node D  │─────▶│  Node C  │
└──────────┘      └──────────┘      └──────────┘      └──────────┘
```

The first node to tell another node that it is 'in the data stream' for a particular destination node is the node where the EUS resides that is establishing a connection to that particular destination node. For example, if node A wants to establish a connection to node C, and node B is the directly connected target node that node A has selected as its best route to node C, node A would tell node B that it is in the data flow for node C.

If node A tells node B it is no longer in the data stream for node C, then node B will tell its directly connected node that was used as the next best step (target node) to node C (node D) that it is no longer in the data stream. The exception to this is if another directly connected node has told node B that it is in the data stream for node C.

Basically, if a node is not in the data stream any more it tells the node it has chosen as a 'target node' that it is not in the data stream any more.

All nodes used in communication between nodes are marked as in the data stream. Being marked in the data stream is not a prerequisite for transferring messages—only having a valid target node is.

A node will maintain a count of the number of directly connected nodes that have told it that it is in the data stream. If the count drops to zero and this node has not marked itself as in the data stream (because an EUS on this node has established a connection to the destination node in question), it will tell its 'target node' for that destination that it is no longer in the data stream.

A node will only tell its target node that it is in the data stream. No other directly connected node will be told that it is in the data stream. A node will ignore a directly connected node that tells it that it is in the data stream if that directly connected node does not tell it that this node is a target node.

## When To Send EUS Messages

Regardless of whether or not a node is marked in the data stream if it receives messages for a destination node that it has a valid a target node for, it will send those messages to the target node.

However, it must follow a series of rules on sending to be sure that the network does not fill up with looping data. These operate separately for each destination node with a valid target node that this node knows about.

1. A node will forward messages to its selected target node (if it has one) for up to X seconds if this node is not marked in the data stream or fHopCost does not equal fHopCostFromFlow. If more then X seconds elapses, this node will delete the EUS messages instead of forwarding them onto the valid target node.
2. If a node has not seen any EUS messages for Y seconds, it will reset the timer used in rule 1. If the timer is reset in rule 1, this node will pass messages for an additional X seconds (as specified in rule 1) before deleting EUS messages.
3. If this node has been marked in the data stream and fHopCost equals fHopCostFromFlow then this node will continue forwarding message for Z seconds after it is no longer marked as in the data stream or fHopCost is not equal to fHopCostFromFlow. After Z seconds it will delete EUS messages sent to it.
4. If this node has been marked in the data stream by a directly connected node and fHopCost equals fHopCostFromFlow then this node will forward all EUS messages it receives to its target node.

This systems works best when $Y < X < Z$. Someone skilled in the art would be able to pick suitable values. An example of values could be:

$Y = 1s$
$X = 3s$
$Z = 4s$

## Sending Destination Node Knowledge

All nodes in the system are ordered by the fHopCostFromFlow value that was sent to it by the selected target node (the hop cost for the connection was added to the fHopCost and fHopCostFromFlow sent by the directly connected node). If

this node is in the data stream (fHopCost == fHopCostFromFlow and sDestNodeUpdate.bIsInDataStream) for a particular destination node we treat the fHopCostFromFlow as 0.

On a per-connection basis we'll treat the fHopCostFromFlow as 0 if this node is in the path of an HSPP for this destination node, and this directly connected node is:

- In the path to the core and the HSPP is a notify HSPP.
- One of the nodes that told us of this HSPP and the HSPP is a request HSPP.

The fHopCostFromFlow will never be infinity, even if the fHopCost is infinity. This is because even if the directly connected node that we have selected as a target node sends us a fHopCostFromFlow of infinity the node will use the last non-infinity value it has instead. It does this so a temporary loss of knowledge won't cause the name of this destination node to be forgotten and cause another destination node to be sent in its place.

A node will ensure that it will not send more then the maximum number of destination nodes requested by the directly connected node. It will try to keep the directly node aware of only the best X destination nodes (based on the effective fHopCostFromFlow) where X is the maximum number of destination nodes request by this directly connected node.

When destination node updates are sent to a directly connected node those with a lower effective fHopCostFromFlow are sent first. The 'effective fHopCostFromFlow' refers to treating destination nodes in the data stream and destination nodes with HSPP's (see above more a more precise definition) as having a fHopCostFromFlow of 0. For example the order in which these different types of destination nodes would be sent:

1. In the data stream
2. In the path of an HSPP (see above for the precise definition)
3. All the rest of the destination nodes ordered by fHopCostFromFlow

If a node wants the directly connected node to 'forget' about a destination node, it will send the directly connected node a route update with a fHopCost and fHopCostFromFlow of infinity (regardless of the actual values). This will allow the directly connected node to forget about that destination node (see above).

A node sends all its 'control messages' in a bandwidth throttled way (discussed later in propagation priorities)

When an update to a destination node route needs to be sent to a directly connected node this destination node is placed in a TreeMap that is maintained for each directly connected node. The TreeMap is a data structure that allows

items to be removed from in by ascending key order. This allows more important updates to be sent to the directly connection node before less important updates.

When sending destination nodes for the first time to a directly connected node, there is no need to place them in the TreeMap for that directly connected node. Updates to destination nodes that have already been sent should be placed in the list for that particular directly connected node so that they can be sent when appropriate. Destination nodes placed in the list should be ordered by their effective fHopCostFromFlow (see above).

The destination node route updates are then sent in this order. When a destination update has been processed it is removed from this ordered list

For example, if there are five destination nodes with effective fHopCostFromFlow values of:

        1.202 - dest node D
        1.341 - dest node F
        3.981 - dest node G
        8.192 - dest node B
        9.084 - dest node M

And the directly connected node has requested a maximum of four destination node routes sent to it, This node will only send the first four in this list (the node will not send the update for destination node M).

If destination node G has its fTempHopCostFromFlow change from 3.981 to 12.231 the new list would look like this:

        1.202 - dest node D
        1.341 - dest node F
        8.192 - dest node B
        9.084 - dest node M
        12.231 - dest node G

In response to this update this node would schedule an update for both destination node G and destination node M. The ordered pairs in the TreeMap for this directly connected node would looks this:

        Position 1-(9.084,M)
        Position 2-(12.231,G)

This node would then send an infinity update for node G. It would then schedule a delayed send for destination node M. (See 'Delayed Sending')

An infinity destination node update has:

        a.   sDestNodeUpdate.fHopCost = INFINITY

    b.  sDestNodeUpdate.fHopCostFromFlow = INFINITY
    c.  sDestNodeUpdate.bIsTargetNode = FALSE
    d.  sDestNodeUpdate.bIsInDataStream = FALSE

When the update for destination node M is sent, it would be non-infinity.


## Delayed Sending

If this is the first time that a destination node update is being sent to a directly connected node, or the last update for that destination node that was sent to this directly connected node had a fHopCost of infinity, the update should be delayed.

For example, if the connection has a latency of 10ms, the update should be delayed by (Latency+1)*2 ms, or in this example 22ms. This latency should also exceed a multiple of the delay between control packet updates (see 'Propagation Priorities). A precise determination of the latency is not important so long as the delay amount exceeds the latency of the connection. If the send delay is significantly more then the latency the network will still operate correctly.

Someone skilled in the art will be able to experiment and find good delay values for their application.

A delay is not needed if:

1. This node is in the path of an HSPP for this destination node, and this directly connected node is:
   a. In the path to the core and the HSPP is a notify HSPP.
   b. One of the nodes that told us of this HSPP and the HSPP is a request HSPP.

This delay is a critical component in the operation of this network.

If an infinity update has been scheduled to be sent (by having it placed in the TreeMap for the directly connected node), but has not been sent by the time a non-infinity update is scheduled to be sent (because it has been delayed), the infinity must be sent first, and then a non-infinity update should be delayed again before being sent.

## Cycling a Destination Node From Infinity to Non-Infinity

If both these criteria are met for this directly connected node:

1. If the maximum number of nodes this directly connected node has asked us to send exceeds or equals the number of destination nodes we're able to send
2. and a destination node with a non-infinity fHopCost moves to end of the list of destination nodes (based on its effective

fTempHopCostFromFlow), or is at the end of the list and has its effective fTempHopCostFromFlow increase even more.

This node will send the directly connected node an update of infinity for this destination node. Then after a suitable delay (See Delayed Send) this node will send a non-infinity update for this destination node to the directly connected node.

This is part of the approach we use to remove bad route data from the network, and automatically remove loops.

An infinity update is an update with the fHopCost value set to infinity (see above for a more complete definition).

The decision to send an infinity update (followed some time later with a non-infinity update for the same destination node) when a destination node moves to the end of the list of all nodes is a recommended approach. Alternative approaches to trigger the infinity update followed by the delayed non-infinity update are:

1. When the fHopCostFromFlow increases by a certain percent, or amount in a specific period of time. For example, if the fHopCost from flow increased by more then 10 times the connection cost in under .5s.
2. When the position of this destination node in the ordered list moves more then (for example) 100 positions in the list, or moves more then (for example) 10% of the list in X seconds.

Someone skilled in the art would be able to determine what a suitable increase in fHopCostFromFlow would be in order to trigger the infinity/non-infinity send.

If a previously unknown destination node appears at the top of the list, the infinity does not need to be sent because the directly connected nodes have not been told a non-infinity update before. However, telling the directly connected nodes about this destination node should be delayed.

This delayed sending does not occur if:

1. This node is in the path of an HSPP for this destination node, and this directly connected node is:
   a. In the path to the core and the HSPP is a notify HSPP.
   b. One of the nodes that told us of this HSPP and the HSPP is a request HSPP.

## Sending Destination Node Updates While In the Data Stream

If the node is in the data stream (because a directly connected node that has picked this node as a target node has told it that it was in the data stream), and fHopCost equals fHopCostFromFlow the node will send the actual fHopCostFromFlow to directly connected nodes that tell this node that it is in the data stream.

Nodes that do not tell this node that it is in the data stream will be sent updates to this destination node with an fHopCostFromFlow equal to .1 or some suitably small number.

This allows updates around the data path to spread more quickly, and speed up the convergence to a more optimal data path.

A node N will tell its directly connected nodes a fHopCost of 0 and a fHopCostFromFlow of .1 (or some suitably small value) for destination node N (itself). The exception to this is if a directly connected node tells node N that it is a target node for destination node N. In this case node N will tell that directly connected node a fHopCostFromFlow of 0.

If this directly connected node tells node N that it is no longer a target node for destination node N, then node N will tell that directly connected a fHopCostFromFlow of .1 for destination node N.


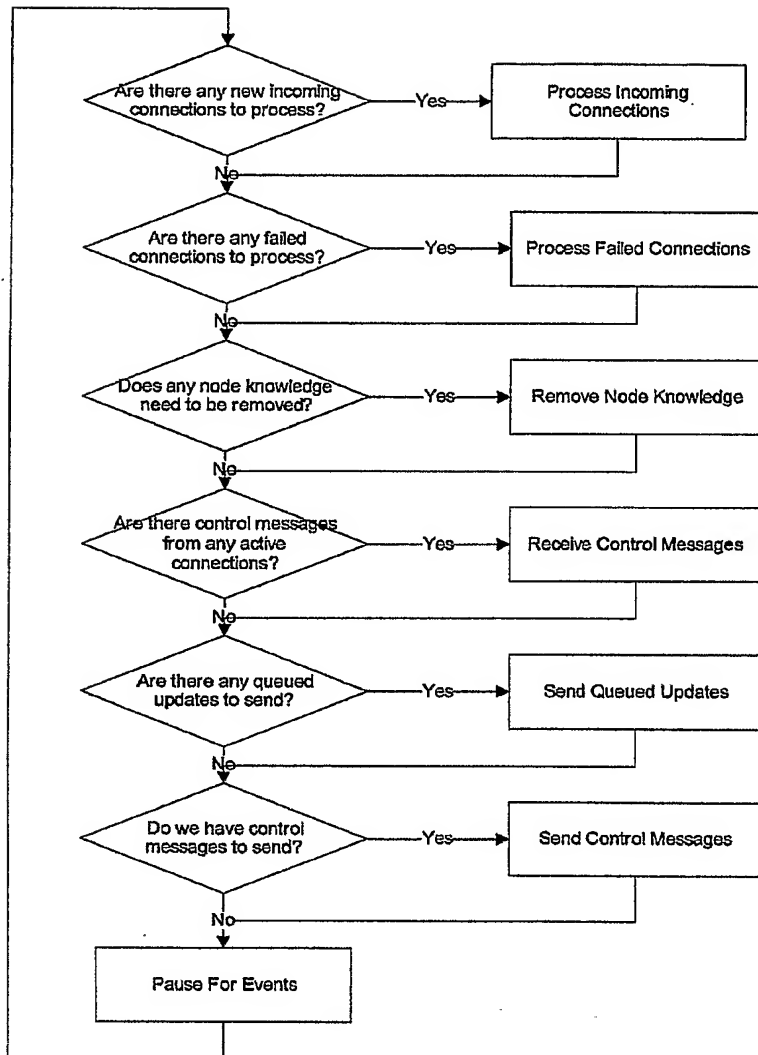## When To Send A Destination Node Update

A destination node update should be sent to directly connected nodes whenever the fHopCost, fHopCostFromFlow, nWeight, InTheDataStream status or target node status changes.

It can happen that an update does not need to be sent to all directly connected nodes because the update would be identical to the last update sent. These duplicate updates can be skipped. However, redundant updates don't affect the correctness of the network, only convergence speed.

If multiple updates need to be sent to a directly connected node, the updates will be sent based on their priority (discussed previously in 'Sending Node Knowledge') and they will be bandwidth throttled (see propagation priorities)


## Basic Command Loop

This is the basic command loop to deal with network control events. Data being switch through the network occurs independently from this command loop.

```
          ┌──────────────────────────────────────┐
          │                  ▼
          │        ◇ Are there any new incoming   ──Yes──▶ ┌─────────────────┐
          │          connections to process? ◇              │ Process Incoming │
          │                  │                              │   Connections    │
          │                  No                             └─────────────────┘
          │                  ▼                                       │
          │        ◇ Are there any failed       ──Yes──▶ ┌──────────────────────────┐
          │          connections to process? ◇            │ Process Failed Connections │
          │                  │                            └──────────────────────────┘
          │                  No                                      │
          │                  ▼                                       │
          │        ◇ Does any node knowledge    ──Yes──▶ ┌─────────────────────────┐
          │          need to be removed? ◇                │ Remove Node Knowledge    │
          │                  │                            └─────────────────────────┘
          │                  No                                      │
          │                  ▼                                       │
          │        ◇ Are there control messages  ──Yes──▶ ┌─────────────────────────┐
          │          from any active                      │ Receive Control Messages │
          │          connections? ◇                       └─────────────────────────┘
          │                  │                                       │
          │                  No                                      │
          │                  ▼                                       │
          │        ◇ Are there any queued       ──Yes──▶ ┌─────────────────────┐
          │          updates to send? ◇                   │ Send Queued Updates  │
          │                  │                            └─────────────────────┘
          │                  No                                      │
          │                  ▼                                       │
          │        ◇ Do we have control         ──Yes──▶ ┌─────────────────────┐
          │          messages to send? ◇                  │ Send Control Messages│
          │                  │                            └─────────────────────┘
          │                  No                                      │
          │                  ▼                                       │
          │        ┌─────────────────┐                               │
          └────────│ Pause For Events │◀──────────────────────────────┘
                   └─────────────────┘
```

## Sending Messages without a Target Node

If node A does not have a target node for messages going to node B it will hold onto those messages in hopes that it will be able to select a valid target node soon.

In the circumstance with no target nodes, and a message needing to be sent, the message will be held onto for X seconds (for example 1s), or until the memory

23

occupied by that message needs to be reclaimed. If the timeout expires, or the memory needs to be reclaimed, the message should be deleted.

If a directly connected node can be selected as a target node, the message will be sent to this directly connected node.

## Connecting Two Nodes Across the Network

The following is an example of one approach that can be used to connect two nodes in this network. This example (like all examples in this patent) is not meant to limit the scope of the patent. Someone skilled in the art would be aware of many variations.

If node A wishes to establish a connection with node B, it will first send out a request HSPP (discussed later) to all directly connected nodes. This request HSPP will draw and maintain route information about node B to node A. This request HSPP will be sent out even if node A already has knowledge of node B.

Once Node A has a valid target node for node B it will send out a 'connection request message' to the specified port on node B. This request will be sent to the directly connected node that has been selected as the target node for messages going to node B.

It will keep sending this message every X seconds (for example 15 seconds), until a sConnectionAccept message has been received, or a timeout has been reached without reception (for example 120 seconds). The connection request message will contain the GUID of node A, and what port to send the connection reply message to. It will also contain a nUniqueRequestID that is used to allow node B to detect and ignore duplicate requests from node A.

The connection request message looks like this:

```
struct sConnectionRequest {

        // the name of node A, could be replaced with a number
        // for reduced overhead.
        sNodeName        nnNameA;

        // Which port on node A to reply to
        int              nSystemDataPort;

        // Which port to send end user messages to on node A
        int              nUserDataPort;

        // a unique request id that node B can use to
        // decide which duplicate requests to ignore
        int              nUniqueRequestID;
```

}

When node B receives the 'connection request' message from node A it will generate a request HSPP for node A and send it to all directly connected nodes. This will draw and maintain route information about node A to node B.

Node B will wait until it has a valid target node (its next best step) for node A , and then mark that target node. as 'In the data stream' for messages destined to node A.

Node B will then send a sConnectionAccept message to node A on the port specified (sConnectionRequest.nSystemDataPort). This message looks like this:

```
struct sConnectionAccept {

        // the name of node B
        sNodeName nnNameB;

        // the port for user data on node B
        int             nUserDataPortB;

        // the unique request ID provided by A in the
        // sConnectionRequest message
        int             nUniqueRequestID;
}
```

The sConnectionAccept message will be sent until node A sends a sConnectionConfirmed message that is received by node B, or a timeout occurs.

When node A receives the sConnectionAccept message it will mark the route to node B as 'in the data stream'.

The sConnectionConfirmed message looks like this:

```
struct sConnectionConfirmed {

        // the name of node A, could be replaced with a number
        // for reduced overhead.
        sNodeName        nnNameA;

        // the unique request ID provided by A in the
        // sConnectionRequest message
        int             nUniqueRequestID;

}
```

If a timeout occurs during the process the connection is deemed to have failed and will be dismantled. The request HSPP's that both nodes have generated will be removed, and the 'in the data stream' flag(s) will be removed.

Once the connection is established, both nodes may send user data messages to each others respective ports. These messages would then be routed to the end user software via sockets (in the case of TCP/IP).

A connection is deemed to have failed when fHopCost is not equal to the fHopCostFromFlow for the ultimate destination node for more then 60 seconds. 60 seconds is an example, someone skilled in the art would be able to pick a suitable value.

A failed connected can also be detected using keep-alive messages, or at the EUS level.

## Node Name Optimization and Messages

Every node update and EUS message need to have a way to identify which destination node they reference. Node names and GUIDSs can easily be long, and inefficient to send with every message and node update. Nodes can make these sends more efficient by using numbers to represent long names.

For example, if node A wants to tell node B about a destination node named 'THISISALONGNODENAME.GUID', it could first tell node B that:

1 = 'THISISALONGNODENAME.GUID'

A structure for this could look like:

```
struct sCreateQNameMapping {
        int             nNameSize;
        char            cNodeName[Size];
        int             nMappedNumber;
};
```

Then instead of sending the long node name each time it wants to send a destination node update, or message - it can send a number that represents that node name (sCreateQNameMapping.nMappedNumber). When node A decides it no longer wants to tell node B about the destination node called 'THISISALONGNODENAME.GUID', it could tell B to forget about the mapping.

That structure would look like:

```
struct sRemoveQNameMapping {
        int             nMappedNumber;
};
```

Each node would maintain its own internal mapping of what names mapped to which numbers. It would also keep a translation table so that it could convert a name from a directly connected node to its own naming scheme. For example, a node A might use:

1 = 'THISISALONGNODENAME.GUID'

And node B would use:

632 = 'THISISALONGNODENAME.GUID'

Thus node B, would have a mapping that would allow it to convert node A's numbering scheme to a numbering scheme that makes sense for node B. In this example it would be:

| Node A | Node B |
|--------|--------|
| 1      | 632    |
| ...    | ...    |
| ...    | ...    |

Using this numbering scheme also allows messages to be easily tagged as to which destination node they are destined for. For example, if the system had a message of 100 bytes, it would reserve the first four bytes to store the destination node name the message is being sent to, followed by the message. This would make the total message size 104 bytes. An example of this structure also includes the size of the message:

```
struct sMessage {
        int             uiNodeID;
        int             uiMsgSize;
        char            cMsg[uiMsgSize];
}
```

When this message is received by the destination node, that node would refer to its translation table to decide which directly connected node this message should be sent to.

These quick destination numbers could be placed in a TCP/IP header by someone skilled in the art.


## When To Remove Name Mappings

If a destination node has a fHopCost of infinity continuously for more then X ms (for example 500 ms) and it has sent this update to all directly connected nodes, then this node will remove knowledge of this destination node.

First it will release all the memory associated with this destination node, and the updates that were provided to it by the directly connected node. It will also

remove any messages that this node has waiting to send to that destination node.

Next it will tell its directly connected nodes to forget the number->name mapping for this destination node.

Once all of the directly connected nodes tell this node that it too can forget about their number->name mappings for this destination node, then this node can remove its own number->name mapping.

At this stage there is no longer any memory associated with this destination node.

A node should attempt to reuse forgotten internal node numbers before using new numbers.

## Simpler Fast Routing

An optimization would be add another column to the name mapping table indicating which directly connected node will be receiving the message:

Thus node B, would have a mapping that would allow it to convert node A's numbering scheme to a numbering scheme that makes sense for node B. In this example it would be:

| Node A | Node B | Directly Connected Node Message is Being Sent To |
|--------|--------|--------------------------------------------------|
| 1 | 632 | 7 |
| ... | ... | |
| ... | ... | |

This allows the entire routing process to be one array lookup. If node A sent a message to node B with a destination node 1, the routing process would look like this:

1. Node B create a pointer to the mapping in question:
   sMapping *pMap = &NodeMapping[pMessage->uiNodeID];
2. Node B will now convert the name:
   pMessage->uiNodeID = pMap->uiNodeBName;
3. And then route the message to the specified directly connected node:
   RouteMessage(pMessage,pMap->uiDirectlyConnectedNodeID;

For this scheme to work correctly, if a node decides to change which directly connected node it will route messages to a directly connected node, it will need to update these routing tables for all directly connected nodes.

If the directly connected nodes reuse internal node numbers, and the number of destination nodes that these nodes know about are less the amount of memory available for storing these node numbers. Then the node can use array lookups for sending messages.

This will provide the node with O(1) message routing (see above). If the node numbers provided by the directly connection nodes exceed size of memory available for the lookup arrays (but the total node count still fits in memory), the node could shift from using an array lookup to using a hashmap lookup.

## More Complex Fast Routing

In order to ensure that nodes can always perform the fast O(1) array lookup routing, a node could provide each directly connected node with a unique node number->name mappings. This will ensure that the directly connected node won't need to resort to using a hash table to perform message routing (see above)

When generating these unique number->name mappings for the directly connected node, this node would make sure to re-use all numbers possible.

By reusing these numbers, it ensures that the highest number used in the mappings should never greatly exceed the maximum number of destination node updates requested by that directly connected node.

For each connection the node will need to create an array of integers, where the offset corresponds to the nodes own internal node ID, and the number stored at that offset is the unique number->name mapping used to for that directly connected node.

The fast routing would then look like this (see above):

1. Node B create a pointer to the mapping in question:
   sMapping *pMap = &NodeMapping[pMessage->uiNodeID];
2. Node B will now convert the name:
   pMessage->uiNodeID = pMap->uiNodeBName;
3. And then route the message to the specified directly connected node:
   RouteMessage(pMessage,pMap->uiDirectlyConnectedNodeID;
4. Before sending, the name will get changed one final time
   pMessage->uiNodeID = UniqueNameMapping[uiConnectionID][pMessage->uiNodeID];

Where UniqueNameMapping is a two dimensional array with the first parameter being the connection ID, and the second is the number used in the unique number->name mapping for the connection with that connection ID.

Reusing the numbers used in the number->name mappings for each directly connected node will require an array that is the same size as the maximum number of mappings that will be used. The array will be treated as a stack with the numbers to be reused being placed in this stack. An offset into the stack will

29

tell this node where to place the next number to be reused and where retrieve numbers to be re-used.

If the directly connected node has requested a maximum number of destination nodes that is greater then the total number of destination nodes known about by this node, then a unique mapping scheme is not needed for that directly connected node.

If this circumstance changes, one can be easily generated by someone skilled in the art.

## Path to Destination Node Removed

If the connection between node A and directly connected node (node B) is broken (or the directly connected node is removed from the network), node A will need to find new 'target nodes' for those destination nodes that were using node B.

Node A will pick the directly connected node with the lowest non-infinity fHopCost that has not picked node A as a 'target node'. If node A is unable to find a directly connected node that matches this criteria, then it will set that destination nodes' fHopCost to infinity. When it sets that destination nodes fHopCost to infinity it will keep the last known fHopCostFromFlow value.

This node will never set a fHopCostFromFlow for a destination node to infinity, even if there are no valid target nodes for that destination node.

Since we use a poison reverse, we'll eliminate most loops immediately. Those loops we don't eliminate will cycle upwards, which in existing approaches will eat up large amounts of network bandwidth and cause the network to stop working.

In our approach we use the value fHopCostFromFlow to determine which updates are sent to which node, and in what order those updates are sent. If a loop is created, there are two possibilites:

Possibility One
The actual destination node has been removed from the network (or can't be connected to from here).

In this case, there is no lower latency that can be used to resolve this loop. However, since fHopCostFromFlow will spiral upwards as quickly as fHopCost, the loop will be very quickly have its update priority set as low as possible, and it will be removed from the list of destination nodes sent to low memory nodes.

In time, the system will detect that a loop is present an remove it. However, time is not pressing, since almost no resources are utilized to support the loop.

Possibility Two
The actual destination node is reachable from here, however lower fHopCost updates from that destination have not reached the loop yet to resolve it.

Initially the loop will cycle up the same way as it does it possibility one, and very quickly render itself irrelevant. If the directly connected nodes are at all memory limited, it will be removed from the list of destination nodes sent to those directly connected nodes.

When the fHopCost that comes from the actually destination node reaches the loop, the loop will be automatically resolved, since the fHopCost will be less that the increasing spiral of fHopCosts in the loop.

If the fHopCost from the actual destination node reaches the loop after the loop has been detected and removed, it will simply act as if the update was spreading to the nodes in the ex-loop for the first time.

This is discussed further in 'Resolving Accidentally Created Loops'.

## Converging on Optimal Paths

A node will select as a 'target node' (for a particular destination node) any node that offers it a lower hop cost then its current 'target node' for a particular destination node. It will never pick a directly connected node that uses this node as its 'target node'. This is the poison reverse in action.

If it is unable to find a target node that has a fHopCost that is non-infinity, and doesn't trigger the poison reverse, then it will set its fHopCost to infinity and keep the last used fHopCostFromFlow.

When a node changes its fHopCost, fHopCostFromFlow, Target node status or in the data stream status, it will schedule an update to be sent to all directly connected nodes.

This process is very similar to Dykstra's algorithm.

## Resolving Accidentally Created Loops

A very useful feature of this network is that stale route data and loops are automatically removed without explicit detection or removal.

3/

There are four mechanisms that allow this to function:

1. Delayed send of destination node route updates to directly connected nodes when this is the first update to this destination node sent to that directly connected node, or the last update had an fHopCost of infinity. Updates with an fHopCost of infinity are sent without delay. (See 'Delayed Sending')
2. If a node has no suitable target node, it will set its fHopCost to infinity and tell all directly connected nodes this new value.
3. A node in the data stream will only send out fHopCosts of 0, if both its fHopCost and fHopCostFromFlow are equivalent. (See 'Sending While in Data Stream)
4. When a node moves to end of the list of all known nodes, or increases its effective fHopCostFromFlow while at the end of this list the destination node is cycled from infinity to non-infinity.

Loops and stale route information have virtually no impact after a short amount of time since their fHopCostFromFlow moves them very quickly out of the priority sends.

Once their fHopCostFromFlow grows to point where it starts to cycle from infinity to non-infinity, the knowledge of that node will quickly be removed.

The rules on when to send EUS messages stop the looping of data in a network.


## Large Networks

In very large networks with a large variation in interconnect speed and node capability different technique need to be employed to ensure that any given node can connect to any destination node in the network, even if there are millions of nodes.

Using the original method, knowledge of a destination node will spread quickly through a network. The problem in very large networks that contain large numbers of destination nodes is three fold:

1. The bandwidth required to keep every node informed of all destination nodes grows to a point where there is no bandwidth left for data.
2. Bandwidth throttling on destination node updates used to ensure that data can flow will slow the propagation of destination node updates greatly.
3. Nodes with not enough memory to hold every destination node will potentially cut off possible ultimate receivers from large parts of the network.

The solution is found by determining what constitutes the 'core' of the network. The core of the network will most likely have nodes with more memory and

bandwidth then an average node, and most likely to be centrally located topologically.

Since this new network system does not have any knowledge of network topology, or any other nodes in the network except the nodes directly connected to it, nodes can only approximate where the core of the network is.

This is done by examining which directly connected node is a 'target node' for the most destination nodes. A directly connected node is picked as a 'target node' because it has the lowest fHopCost, and the lowest fHopCost will generally be provided by the directly connected node that is closest to the ultimate destination node. If a node is used as a 'target node' for more destination nodes then any other directly connected node, then this node is probably a step toward the core of the network.

If there is a tie between a set of directly connected nodes for who was picked as the 'target node' for the most destination nodes, the directly connected node with the highest 'tie-breaker' value (which was passed during initialization) will be selected as the next best step towards the core. This mechanism will ensure that there are no loops in non-trival network (besides Node A to Node B to Node A type loops).

When a node has chosen a directly connected node as its 'next best step to the core' it will tell that directly connected node of its choice. This allows nodes to detect when they have generated a core that no other nodes are using as their core. This is explained in detail later. The message that is passed looks like this:

```
struct sCoreMessage {
        bool            bIsNextStepToCore;
}
```

Since nodes not at the core of the network will generally not have as much memory as nodes at the core, they may be forced to forget about an ultimate receiver that relies on them to allow others to connect. If they did forget, no other node in the network would be able to connect to that ultimate receiver.

In the same way, a node that is looking to establish a connection with an ultimate receiver faces the same problem. The destination node definition that it is looking for won't reach it fast enough - or maybe not at all if it is surrounded by low capacity nodes.

The solution to these problems is to set up a high speed propagation path (HSPP) between the node that is the receiver or sender to the core of the network. A HSPP is tied to a particular destination node name or class of destination node names. If a node is in a HSPP for a particular destination node it will immediately process and send:

1. Initial knowledge of the destination node
2. When the destination node fHopCost goes to infinity
3. When the destination node fHopCost moves from infinity to some other value

To those nodes directly in the HSPP. This will ensure that all nodes in the HSPP will always know about the destination node in question, if any one of those nodes can 'see' the destination node.

Node knowledge is not contained in the HSPP. The HSPP only sets up a path with a very high priority for knowledge of a particular destination node. That means, that any destination node update that is in one of the previous three categories will be immediately sent.

There are two types of HSPP's. One type pushes knowledge of a destination node towards the core, the second pulls knowledge of a destination towards the node that created the HSPP.

When an destination node is first connected to the network, it will create an HSPP based on its node name. This HSPP will be of the type that pushes knowledge of this node towards the core of the network. The HSPP created by the ultimate destination node will be maintained for the life of the node. If the node is disconnected, the HSPP will be removed.

If an ultimate sender is trying to connect to an ultimate destination node it will create a request HSPP. This HSPP is the type that will pull knowledge of the destination node back towards the node that wants to connect to the destination node.

Both types of HSPP will travel to the core. The HSPP that sends knowledge of the node to the core will be maintained for the life of the node. The request HSPP will be maintained for the life of the connection.

An HSPP is not a path itself, rather it forces nodes on the path to retain knowledge of the node in question, and send knowledge of that node quickly along the path.

The HSPP does not specify where EUS messages flow. The HSPP is only there to guarantee that there is always at least one path to the core.

The nodes that generate the HSPP will always send that HSPP to all directly connected nodes.

A node will not send a directly connected node more HSPP's then the maximum node code requested by that directly connected node.

An HSPP does not specify where data should flow, it only guarantees a connection (possibly non-optimal) between nodes, or one node and the core.

34

## Detecting an isolated core

A core is defined as two directly connected nodes that have selected each other as the next best to the core.
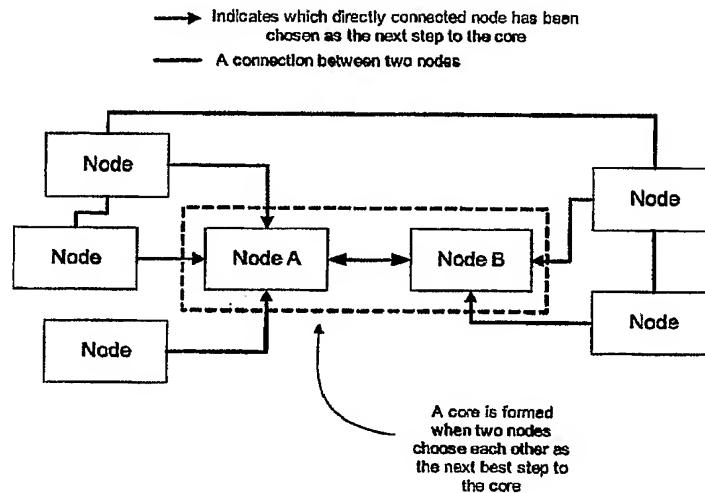


If a core is created, both nodes that form the core (in this example Node A and Node B) will check to see how many directly connected nodes they have. If there is more then one directly connected node then they will examine all the other directly connected nodes.

If the only directly connected node that has chosen this node as its next best step to the core is the node that has caused the core to be created, then this node will select its next best choice to be the next best step to the core.

This will eliminate cores that will block the flow of knowledge to the real core.

## Alternative Ways to Select the Next Best Step to the Core

The approach discussed previously involved assigning a credit of '1' to a directly connected node for each destination node that selects that directly connected node as a target node. The node with the highest count is the next best step to the core (or in the case of an isolated core, the second highest count).

Instead of assigning a credit of one to each directly connected node for each destination node that selects it as its best choice, other values can be used.

3ʃ

For example, log(fHopCost+1)*500 can be the credit assigned. Other metrics could also be used. This metric has the advantage of giving more weight to those destination nodes that are further away. In a dense mesh with similar connections and nodes, this type of metric can help better, more centralized cores form.

Another approach, which can be used to extend the idea of providing more weighting to destination nodes that are further away is to order all destination nodes by their hop costs, and only use the x% (for example, 50%) that are the furthest away to determine the next best step to the core.

The nWeight value (that is in the sDestNodeUpdate) can also be used to help cores form near more powerful nodes. For example the credit assigned could be multiplied by $10^{nWeight}$ (where 10 is an example).

This will help cores form near the one or two large nodes, even if they are surrounded by millions of very low power nodes.

The nWeight value should be assigned in a consistent fashion across all nodes in the network. Possible nWeight values for types of nodes:

| nWeight value | Type of Node | Equivliant to X low capacity sensors |
|---|---|---|
| 0 | A very low capacity sensor or mote | 1 |
| 1 | A bigger sensor | 10 |
| 2 | A bigger sensor with more battery life and memory | 100 |
| 3 | A cell phone | 1000 |
| 10 | A home computer | 10000000000 |
| 15 | A core router | 1000000000000000 |
| 20 | A super computer with massive connectivity and memory | 100000000000000000000 |

These weight values are suggestions only, someone skilled in the art would be able to assign suitable values for their application.

## How an HSPP is Established and Maintained

If a node is told of an HSPP it must remember that HSPP until it is told to forget that HSPP, or the connection between it and the node that told it of the HSPP is broken.

Each node must store as many HSPP's as are given to it. A node should not send more HSPP's to a directly connected node then the maximum destination node count that directly connected node requested.

In most systems the amount of memory available on nodes will be such that it can assumed that there is always enough memory, and that no matter how many HSPP's pass through a node it will be able to store them all. This is even more likely because the number of HSPP's on a node will be roughly related to how close this node is to the core, and a node is usually not close to a core unless it has lots of capacity, and therefore probably lots of memory.

UR = Ultimate Receiver
US = Ultimate Sender

An HSPP takes the form of:

```
struct sHSPP {

        // The name of the node could be replaced with a number
        // (discussed previously)
        sNodeName     nnName;

        // a boolean to tell the node if the HSPP is being
        // activated or removed.
        bool          bActive;

        // a boolean to decide if this a UR (or US generated HSPP)
        bool          bURGenerated;

};
```

It is important that the HSPP does not loop back on itself, even if the HSPP's path is changed or broken. This is guaranteed by the process in which the next step to the core of the network is generated.

A node will never send an HSPP back to a node that has sent it an active HSPP of the same name.

A node will record the number of directly connected nodes that tell it to maintain the HSPP (bActive is set to true in the structure). If this count drops below zero it will tell its directly connected node that is the next best step to the core to forget about the HSPP (bActive is set to false in the structure).

At a broad level we're trying to allow the HSPP to find a non-looping path to the core, and when it reaches the core, we want to stop spreading the HSPP. If the HSPP path is cut, the HSPP from the cut to the core will be removed.

The purpose of the HSPP generated by the UR is to maintain a path between it and the core at all times, so that all nodes in the system can find it by sending a US generated HSPP (a request HSPP) to the core.

37

If multiple HSPP's for the same destination node arrive at the same node, that node will send on an HSPP with bURGenerated marked as true, if any of the incoming HSPP's have their bURGenerated marked as true.

If the directly connected node that was selected as the next best step to the core changes from node A to node B, then all the HSPP's that were sent to node A will be sent to node B instead. Those HSPP's that were sent to node A will have their 'bActive' values set to false, and the ones sent to node B will have their 'bActive' values set to true.

The HSPP that is generated by a node to drive knowledge of itself to the core should be sent to all directly connected nodes. This ensures that even if this node is moving rapidly, that knowledge of it is always driven to the core.

When a node A establishes a connection to another node B, Node A uses an HSPP to pull route information for node B to itself (called a request HSPP). This HSPP should also be sent to all directly connected nodes.

## Propagation Priorities

Bandwidth throttling for control messages will need to be used.

Total 'control' bandwidth should be limited to a percent of the maximum bandwidth available for all data.

For example, we may specify 5% of maximum bandwidth for each group, with a minimum size of 4K. In a simple 10MB/s connection this would mean that we'd send a 4K packet of information every:

$$= 4096 / (10MB/s * 0.05)$$
$$= 0.0819s$$

So in this connection we'd be able to send a control packet every 0.0819s, or approximately 12 times every second.

The percentages and sizes of blocks to send are examples, and can be changed by someone skilled in the art to better meet the requirements of their application.

## Bandwidth Throttled Messages

These messages should be concatenated together to fit into the size of block control messages fit into.

If a control message references a destination node name by its quick-reference number, and the directly connected node does not know that number, then quick reference (number->name mapping) should precede the message.

35

There should be a split between the amount of control bandwidth allocated to route updates and the amount of control bandwidth allocated to HSPP updates.

For example, 75% of the control bandwidth could be allocated to route updates and the remaining 25% could be allocated to HSPP updates. Someone skilled in the art could modify these numbers to better suit their implementation.

## Information Structures For Each Connection

Each connection to a directly connected node needs the following information stored:

1. Connection Handle
   A Socket handle, or way to access the connection between this node and the directly connected node.
2. Tie Breaker Value
   This value was sent during the initial connection process and is used to determine who wins in the event of a tie.
3. Number To Name Mappings
   The directly connected node will use a numbering scheme that will be different from this nodes numbering scheme. This mapping allows this node to convert the numbering scheme of the directly connected node the numbering scheme of this node.
4. Was A Non-Infinity Update Sent For a Destination Node
   It is important that the node know if the last fHopCost update for a particular destination node was non-infinity. If it was an infinity update, then the next non-infinity update may need to be delayed before being sent.
5. Was the number->name mapping sent?
   Has this node sent its own number->name mapping to this directly connected node? If we haven't then we'll need to send this mapping before we sent any messages that reference this mapping.
6. The last destination node update
   When a directly connected node sends us an update (sDestNodeUpdate) we'll store the contents of that update. Before the update is stored we'll adjust it to reflect the connection cost of this connection.
7. Is this node the next core step
   Has this directly connected node told us that this node is its next step towards the core
8. Maximum destination nodes requested
   During the initial connection process this directly connected node told us the maximum number of destination nodes that it wants to be told about.
9. Connection Cost
   The connection cost that has been assigned to this connection. This

value will be added to the fHopCost and fHopCostFromFlow of updates received from this directly connected node

10. <u>Destination Node Update List</u>
This is a list of destination nodes that are scheduled to be sent to this connection. See 'Sending Node Knowledge'

11. <u>Which HSPP's this node has received</u>
If an active HSPP has be received from this directly connected node we'll record if it was a Notify or Request HSPP.

12. <u>Which HSPP's this node has sent</u>
The type of active HSPP that has been sent to this directly connected node.

This is not an exhaustive list.

40

**I Claim:**

1. A system for transmission of messages between nodes on a network, said system comprising:

   (a) a plurality of destination node knowledge on each node; and
   (b) a network communication manager on each node, wherein said network communication manager has knowledge of neighbour nodes and knowledge of all queues on each node

2. A method for determining the best path through the network comprising the steps of :
   (a) Determining the hop cost of neighbour nodes and selecting the most efficient neighbour node to receive a message; and
   (b) Repeating step (a) on a regular basis